C/C++ Contributing Editor

The New C: Compound Literals

Randy Meyers

Structs in C are not quite “first class types,” but with the help of compound literals, they are at least a lot easier to use.

The creators and critics of programming languages sometimes classify the data types in a programming language as to whether they are first class types or not. A first class type is one that has the full set of reasonable operations and possible uses defined for it. For example, arrays in C are not first class types because you cannot perform array assignment using the assignment operator or pass an entire array by value as an argument or return an array as a result from a function. In contrast, **int** in most programming languages is the quintessential first class type: Not only are all reasonable operators defined upon **int**, but you can also have arrays of **int**, pass **int** as an argument, return **int** from a function, and so on.

Originally, structs in C suffered many of the same deficiencies as arrays, but it was commonplace even before the ANSI C Standard for compilers to support struct assignments, struct arguments, and struct function return values. Structs in modern C are almost first class types, but they still lack support for comparisons for equality or inequality using the == and **!**= operators. The C committee has entertained proposals for supporting == and **!**= for structs, but the debate over how to treat union members of structs caused the proposal to be shelved.

You might wonder, if structs need the equality operators defined in order to be first class types, do they also need the relational operators, e.g. < or >, to be defined in order to be first class types? This brings us to the “reasonable” in the definition of first class types above. Consider:

```
struct S {int a, b;};
struct S x = {1,2};
struct S y = {2,1};
```

Given that **x.a** < **y.a** but **x.b** > **y.b**, is it more reasonable to say that **x** < **y**, or that **x** > **y**, or that no automatic, general definition of < and > on structs is reasonable? I would argue that since programmers lay out structs in order to minimize padding, or to match an externally declared layout, or in the order that members occur to them, and not in an order that results in a natural comparison order for < and >, that it is unreasonable to provide a standard definition of < and > in the C Language.

Not surprisingly, students of programming language design at times disagree whether a particular operation or use of a type is reasonable or necessary in order to be a first class type. Dennis Ritchie pointed out [1] that some might not consider structs in C90 to be first class types because there are no constants of type struct.

C99 [2] added exactly that feature: constants of almost any type including struct, union, and array. This feature, called compound literals, is based on the brace-

enclosed initializer syntax. The motivation for adding this feature to C99 was its notational conciseness, convenience, and usefulness, rather than an abstract desire to make struct a first class type.

Constant Versus Literal

Compound literals are not true constants in that the value of the literal might change, as is shown later. This brings us to a bit of terminology. The C99 and C90 Standards [2, 3] use the word "constant" for tokens that represent truly unchangeable values that are impossible to modify in the language. Thus, **10** and **3.14** are an integer decimal constant and a floating constant of type **double**, respectively. The word "literal" is used for the representation of a value that might not be so constant. For example, early C implementations permitted the values of quoted strings to be modified. C90 and C99 banned the practice by saying that any program than modified a string literal had undefined behavior, which is the Standard's way of saying it might work, or the program might fail in a mysterious way. This allowed implementations to pool strings and place them in read-only storage. However, the Standard knew that some implementations might continue allowing quoted strings to be modified (sometimes a compiler option must be used), and called tokens like **"ABC"** string literals rather than string constants. Unfortunately, the C++ Standard [4] does not use the word "literal" with the same meaning as the C Standard. In C++, **10** is called an integer literal, for example.

Compound literals might or might not be constant depending upon whether their programmer-specified type is const or not. Unlike string literals, it is portable to modify a non-const compound literal.

Compound Literals

Syntactically, a compound literal looks like a cast followed by a brace-enclosed initializer. Given the following two types:

```
struct POINT {int x, y;};
union U {float f; int i;};
```

Here are some examples of compound literals:

```
(int) {1}
(const int) {2}
(float[2]) {2.7, 3.1}
(struct POINT) {0, 0}
(union U) {1.4}
```

The value of the compound literal is an anonymous object whose type is specified by the "cast." The anonymous object has been initialized by the brace-enclosed initializer list. As the last three compound literals in the above example show, compound literals give you a constant-like notation for arrays, structs, unions, as well as any other object type (except for C99 variable length arrays).

A compound literal can be used anywhere an object with the same type of the compound literal could be used. For example,

```
int x;
x = (int) {1} + (int) {3};
```

is equivalent to

```
int x;
int unnamed1 = {1};
int unnamed2 = {3};
x = unnamed1 + unnamed2;
```

Compound literals are particularly useful as function arguments. For example, suppose you were using a graphics library that used **struct POINT**s to express coordinates. You might draw a pixel in a window like this:

```
extern drawpixel(struct POINT where);
drawpixel((struct POINT) {5, 5});
```

Compound literals yield lvalues. This means that you can take the address of a compound literal, which is the address of the unnamed object declared by the compound literal. As long as the compound literal does not have a const-qualified type, you can use the pointer to modify it.

```
struct POINT *p;
p = &(struct POINT) {1, 1};
p->x = 2;
p->y = 2;
printf("*p = %d, %d\n", p->x, p->y);
```

causes ***p = 2, 2** to be printed.

Compound literals are in effect declarations and initializations of unnamed objects that can appear in expressions. The unnamed objects and their initializations follow the same rules [5] as normal declarations, and have the same special treatment depending upon whether the compound literal appears within a function body or not.

If a compound literal appears outside of a function body, then the unnamed object has static storage duration, just like all other objects declared outside of a function. It is allocated and initialized once before the program begins to run and remains allocated as long as the program is running. Since the initialization occurs before running the program, all of the initializers in the brace-enclosed list must be constant expressions [5].

If a compound literal appears inside the body of a function, then the unnamed object has automatic storage duration and acts like a local variable of the immediately enclosing block. It is allocated and initialized when its “declaration” is reached in the block and deallocated upon exiting the block [5]. The expressions in the brace-enclosed initializer list can be any run-time expressions.

```
void f()
{
  int *p;
  extern int g(void);
  {
    p = &(int) {g()};
    *p = 1;   //OK
  }
  // p points to deallocated
```

```
  // stack space
  *p = 2;   //BAD
}
```

In the same way that the declaration and initialization of an automatic variable acts like an assignment to that variable [5], every time control passes through the body of a compound literal with automatic storage duration, the unnamed variable is reinitialized. Thus, the following function draws a diagonal line from **(0,0)** to **(9, 9)**.

```
void line()
{
  int i;
  for (i = 0; i < 10; ++i)
    drawpixel((struct POINT) {i, i});
}
```

The brace-enclosed initializer list for a compound literal has the same semantics as a brace-enclosed initializer list in a declaration. If you only provide initializers in the list for some of the members of a struct or elements of an array, the other members or elements are implicitly initialized with zeros of the appropriate type. Thus, **(int [10]) {0}** is an array of ten integers all initialized to zero. This means that it might be safer to assign to a struct using a compound literal rather than assigning its members individually. Contrast the following lines in a function:

```
struct POINT p;
p.x = x;
p.y = y;
```

versus:

```
struct POINT p;
p = (struct POINT) {x, y};
```

Suppose in the future you add a **z** member to **POINT** to make it a three-dimensional point. When you assign the members individually, the **z** member never receives a value and contains stack trash. When a compound literal is used to assign **p**, the **z** member is assigned the default value of zero (probably a reasonable default for a 3-D graphics package).

Like any other brace-enclosed initializer list, the initializer list in a compound literal may use the new C99 feature of designated initializers [5], where the member or array element being initialized may be named. When a function takes a struct as an argument, compound literals and designated initializers can be used to call the function with a poor man's version of keyword arguments to a function and default argument values for a function, as in:

```
drawpixel((struct POINT) {.y=12});
```

Here, the designated initializer **.y** acts like a keyword argument to the function, and the **.x** "argument" to the function receives a default value of zero.

Like normal declarations, if the type inside of the "cast" of a compound literal is an array of unknown size, then the number of elements of the array is determined by the brace-enclosed initializer. A compound literal with type array has the same semantics as a variable with type array. Except when used as the operand of **sizeof** or **&**, an

array used in an expression is converted to a pointer to the first element of the array. In the following, **p** points to the first element of an array of three **int**s.

```
int *p;
p = (int []) {1, 2, 3};
```

Normally, every compound literal that you write results in a distinct unnamed object. However, if the type of the compound literal is const-qualified, and the compound literal is initialized with constant expressions, then the compiler is free to pool the compound literals (only store one copy) and to place the unnamed object(s) in write-locked storage. Such compound literals are true constants, not just literals.

Thus, those programmers who worry about whether their types are first class types and consider "having a constant representation" to be a requirement, have one less thing to worry about.

References

[1] Dennis Ritchie. "The Development of the C Programming Language," in Bergin and Gibson, editors, *History of Programming Languages* (Addison Wesley, 1996).

[2] *ANSI/ISO/IEC 9899:1999, Programming Languages - C. 1999.* Available in Adobe PDF format for $18 from <http://www.techstreet.com/ncitsgate.html>.

[3] *ANSI/ISO/IEC 9899:1990, Programming Languages - C. 1990.*

[4] *ANSI/ISO/IEC 14882:1998., Programming Languages - C++. 1998.* Available in Adobe PDF format for $18 from <http://www.techstreet.com/ncitsgate.html>.

[5] Randy Meyers. "The New C: Declarations and Initializations," *C/C++ Users Journal,* April 2001.

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*